Docket 490334.016

METHOD AND SYSTEM FOR DYNAMIC FLOWING DATA TO AN ARBITRARY PATH DEFINED BY A PAGE DESCRIPTION LANGUAGE

CROSS-REFERENCE TO RELATED APPLICATIONS

This application claims priority from U.S. Provisional Application No. 60/107,583 filed November 9, 1998.

BACKGROUND

The present invention relates to the high speed printing industry, and more particularly a system and method for flowing variable data into a page description language file in a high speed printing environment.

Application programs, such as word processors, illustrators, and computer-aided design systems are software packages used to create a document (text and graphics) on a computer screen and to simultaneously generate a page description language ("PDL") specification, which is to be transferred to the printer or to any other type of raster or output device for creating a hard copy or copies of the document. Alternatively, a PDL specification can be generated by a programmer without the assistance of an application program.

The printer executes the PDL specification to generate a bitmap of the document, or a raster-data representation of a document, and eventually transfers the bitmap or raster-data to the physical medium. A typical PDL language, such as PostScript (a registered trademark of Adobe Corporation) defines a page of the document as containing a number of data areas, where each data area contains either graphic or alpha-numeric data. Each data area is defined by a "graphic state," which is a collection of parameters for controlling the representation and appearance of text and graphics. For example, the graphic state can include a set of text attributes such as scale-factor, type-font, etc. In PostScript, an example of a PDL command used to build a graphic state can be: `twenty rotate \Times-Roman findfont 14 scalefont` and `setfont`. Examples of PDL commands used to define the graphic or

alpha-numeric data that is displayed in the data area include: `0 0 moveto` and `(ABC) show`. The entire group of PDL commands used to define a document is hereinafter referred to as the "PDL specification."

In variable data printing each printed document shares a common template and there is at least one area in the template that changes for each printing of the template. Typical PDL languages are not designed for high-speed variable data printing because, with PDL languages and PDL interpreters, even if a single item of data in the document changes, an entirely new PDL specification must be created and interpreted. For example, if one-hundred thousand copies of a mass-mailing advertisement were to printed (i.e., each copy of which is identical except for the mailing address), it is typically necessary to generate a new PDL specification for each copy to printed. Hence, to generate one-hundred thousand advertisements, it would be necessary to generate one-hundred thousand PDL specifications, even though each advertisement is virtually the same except for the variable data area. The processing time required to interpret and render one-hundred thousand PDL specifications is enormous, significantly slowing the entire printing system.

Furthermore, typical PDL languages do not include any text or data flowing capabilities. These features are usually implemented by the application program, and when such an application program flows data (such as text) into a PDL document, the calculations to determine where to place the data are completed prior to the generation to the PDL specification. Accordingly, variable data cannot be flowed into a template document without creating a new PDL specification for each document. Accordingly, there is a need for a high-speed printing operation having the ability to merge variable data into a template defined by a PDL specification; and in particular, having the ability to flow variable data into a template path defined by PDL specification in a high-speed printing operation.

SUMMARY

It is an object of the present invention to provide a system and method for flowing variable data (such as text data, image data, bar code data and the like) into a path of a template defined by a PDL specification in a high-speed printing operation. It is a further object of the present invention to provide the ability to generate a plurality of merged bitmaps, which are each essentially a copy of a template, except for at least one portion of the template that contains an arbitrary path. In that path, each merged bitmap can contain a different set of variable data merged into it. The template is defined by a page description language, and the page description language only needs to be processed or interpreted once before creating all of the merged bitmaps, thus providing an extremely high-speed variable data printing operation.

The computer implemented method for flowing data into an arbitrary path defined by a page description language specification ("PDL specification") generally comprises the steps of: processing (interpreting) the PDL specification to produce a template; designating a path defined in the PDL specification as a wrapping path; associating a block of variable data with the wrapping path; and merging variable data, according to the path boundary and according to a predefined flow rule, into a copy of the template.

The method of the present invention is accomplished by executing a control task in conjunction with a PDL interpreter program. The control task generates a template display list based upon the PDL commands in the PDL specification. The display list includes a plurality of rendering commands, where each rendering command designates a particular data area or object to be rendered, the graphics state to be applied to the data area and the offset address at which the rendered object, if any, in the data area is to be overwritten onto the final bit map. The graphic states for each data area are set forth in the PDL specification, and pertain to the print attributes that describe how particular graphic or alpha-numeric data is to appear on the printed page. These attributes can include the size, font, position, orientation, location, and the like.

The control task, during the PDL interpretation procedure, monitors the data areas defined by the PDL specification to watch for variable data paths defined by the

PDL code. If the control task identifies a path as being a variable data path, it reserves the graphic states associated with that variable data path in a cache or memory, and then moves on to the next data area defined in the PDL specification, preferably without allowing the path data to be added to the template display list.

Once the interpreter program completes its interpretation of the PDL specification, the control task saves the template display list in memory without dispatching a bitmap of the template to the printer. Subsequently, a merge task is initiated which accesses a variable data record from a merge file; associates the variable data record to a particular variable data path; creates representations of the variable data, such as rendering commands according to the reserved graphic states pertaining to that particular variable data path, according to the boundary of the particular variable data path and according to a predefined flow rule; and then generates a merged bitmap by processing the template display list and the variable data rendering commands. The final merged bitmap that may then be dispatched to the printer. This merge task is repeated for each variable data record in the merge file associated with that particular variable data path to create a plurality of the merged bitmaps.

Thus, the PDL specification of the template need only be interpreted once, saving significant processing time for the variable printing operation, because the reserved graphic states may be utilized over and over again to create the flowed data bitmap for each variable data record contained in the merge file.

How the control task identifies a particular PDL path defined in the PDL specification as being unique, i.e., as being identified as a wrapping path, is an important step in the above process. This is accomplished by providing a text command in the PDL specification that defines one or more characters that are recognized by the control task as being special characters, as opposed to merely being characters that are to be included on the printed page. The control task monitors all text strings defined by the PDL specification for such special characters, and responsive to a detection of the special character in the text string defined by the text command, the control task identifies the path command that has a predetermined

relationship with the text command in the PDL specification. This predetermined relationship can be satisfied by the first path command to follow the text command in the PDL specification or by the path command that is "grouped" with the text command in the PDL specification.

In the preferred embodiment of the present invention, the characters "<<" and ">>" are used as part of a special text string to define an area as a variable data area. And if that special text string also includes the string *wrap* then the control task will recognize that the very next path command appearing in the PDL specification will be a unique path, in this case a path for flowing variable text bitmaps into.

BRIEF DESCRIPTION OF THE DRAWINGS

Fig. 1 is a is a schematic, block-diagram representation of a high-speed printing system according to the present invention;

Fig. 2 is a first example of a job ticket file for use with the present invention;

Fig. 3 is a second example of a merge file for use with the present invention;

Fig. 4 is a graphical representation of data contained in a first example PDL specification for use with the present invention;

Fig. 5 is a graphical representation of a process step of the present invention operating on data contained in the PDL specification of Fig. 4;

Fig. 6 is a graphical representation of a process step of the present invention following the process step of Fig. 5;

Fig. 7 is a graphical representation of a process step of the present invention following the process steps of Figs. 5 and 6;

Fig. 8 is a graphical representation of a process step of the present invention following the process steps of Figs. 5 and 6;

Fig. 9 is an example of a merged document created by the process and system of the present invention;

Fig 10 is an example of a merged document created by the process and system of the present invention;

Fig. 11 is a flow chart representation of a process of the present invention;

Fig. 12 is an example of a merged document created by the process and system of the present invention;

Fig. 13 is an example of a merged document created by the process and system of the present invention;

Fig. 14 is a second example of a job ticket file for use with the present invention;

Figs 15A and 15B are a second example of a merge file for use with the present invention;

Fig. 16 is a graphical representation of data contained in a second example PDL specification for use with the present invention;

Figs. 17A-17C are graphical representations of process steps of the present invention operating on data contained in the PDL specification of Fig. 16, the job ticket of Fig. 14 and the merge files of Figs. 15A-15B; and

Figs. 18A-18C are examples of merged pages created by the process of the present invention using the PDL specification of Fig. 16, the job ticket of Fig. 14 and the merge files of Figs. 15A-15B.

DETAILED DESCRIPTION

As shown in Fig. 1, a system for performing the method of the present invention includes a printer controller 10 having access to a job ticket file 12, a page description language ("PDL") file 14, a source of variable data such as a merge file 16, and an optional printer configuration file 18. The system also contains an operator control terminal 20 for providing operator controls such as indicating the name and path of the job ticket file 12 for the specific print job.

The job ticket file 12 contains the guidelines for the print job which can include the names and locations of the PDL file(s) 14, the merge file(s) 16, the configuration file(s) 18, etc.; and may also include special instructions pertaining to features such as data wrapping, described below. The PDL file 14 is preferably a PostScript® (registered TM of Adobe Systems, Inc.) specification created by an application program, such as a word processor, illustrator, or computer-aided design system. The merge file 16 contains platform independent data, such as text data, image data, bar-code data

and the like, which is to be merged into a template bitmap defined by the PDL file during the merging task, as will be described in detail below. The configuration file 18 defines the print engines and post processing equipment and other options to be executed.

Initially, the path and name of the job ticket file 12 is specified by the operator using the operator control terminal 20. The printer controller 10 retrieves the job ticket file 12 and then retrieves the PDL files 14 and merge files 16 that are specified in the job ticket file. Next the controller 10 initiates a control task 22 in conjunction with a page description code interpreter program.

The control task interprets the PDL specification from the PDL file 14 and monitors data areas defined in the PDL specification to watch for areas defined by the specification to become variable. If the control task identifies a data area as being a variable data area, it reserves the graphic states 23 of that variable data area in memory 24 and then moves on to the next data area defined by the PDL specification, usually without allowing any data defined by the variable data area to be added to the template bitmap. Preferably, the control task 22 will also create a font cache (an entire set of character bitmaps generated according to the reserved graphic states) for the reserved graphic states, which will be linked to the reserved graphic states in memory 24. Once the control task completes its processing of the PDL specification, the control task saves the template bitmap 25 in memory 26.

The control task 22 may also create a template display list 25 of static data defined by the PDL file 14. The display list 25 will include a plurality of rendering commands, where each rendering command designates a particular static data area or object to be rendered, the graphics state to be applied to the static data area and the offset address at which the rendered object, if any, in the static data area is to be overwritten onto the final bit map. As mentioned above, the graphic states for each data area are set forth in the PDL specification, and pertain to the print attributes that describe how particular graphic or alpha-numeric data is to appear on the printed page. Once the control task completes its processing of the PDL specification, the control task

may save the template display list 25 in memory 26. If the PDL file 14 does not include code for any static data, the control task may generate an empty template display list 25 or may decide not to create a template display list at all.

Next, a merge task 28, having access to the variable data records 17 from the merge file 16, is executed to apply the reserved graphics states 23 and associated font cache to the variable data records 17, creating rendering commands for that variable data record as defined by the graphic states. The merge task 28 retrieves a copy 25' of the template display list 25 from the memory 26 and merges the variable data rendering commands with the template display list to create a merged display list 30. Finally, the controller 10 performs a rendering task 32 to render the merged display list 30 into a plurality of bitmap bands 34 for dispatching to at least one print engine 36.

A method for performing the above control task and merge task is described in U.S. Pat. No. 5,729,665 entitled "Method of Utilizing Variable Data Fields with a Page Description Language," the disclosure of which is incorporated herein by reference. A method and system architecture for performing the above merging, banding and dispatching operations are respectively described in U.S. Pat. Nos. 5,594,860 5,796,930, the disclosures of which are also incorporated herein by reference.

A first embodiment of the present invention is illustrated by way of example in Figs 2-10. As illustrated in Fig. 2, the job ticket file 12 can contain a file path 38 for determining the location and name of the PDL file, and can contain a file path 40 for determining the location and name of the merge file. The job ticket file 12 can also contain a descriptive name of a path 42, in this case, named "Shape," for identifying a name of a path in the PDL file that is to have variable data flowed into it during the merge task. The variable data to be flowed into the path, text data in this case, will be taken from the file designated by the file path 40 of the merge file. In this case the merge file is named "info.text." The group header 44 "[Wrap]" indicates that the group is defining a wrapping path. After the wrapping path "Shape" has been defined in the job ticket file, a second group header 46 "[Shape]" can be thereafter defined in the job ticket file to provide information about the wrap path; such as defining the fill rule 48 to

be used in the wrapping operation, and such as defining a path drawing rule 50, i.e., whether the path is to be drawn in the final rendered image. Other definable wrapping commands for the particular path "Shape" can include defining the top, bottom or side margins, defining the justification, setting the number of paths to flow the data into, defining an overflow path, etc. A complete description of the different elements that can be defined for the wrapping path in the job ticket file is described in detail in the Appendix, below.

As illustrated in Fig. 3, the merge file 16 is a platform-independent data file that contains the "variable" data to be merged into the path defined in the PDL specification. The merge file can contain a field name 52, corresponding to a field name that will be defined in the PDL specification, which is associated with a particular variable data path. The merge file will also contain a number of variable data blocks 54, text blocks in this case, corresponding to the field name 52. One variable data block 54 will be merged into the variable data path, defined in the PDL specification, at a time.

As illustrated in Fig. 4, the designer will utilize an application program to create a document containing a path 56 and attribute data, such as an attribute string 58, to be associated with the path 56. The application program will then be directed to create a PDL specification of the document by the designer. The attribute string 58 contains a field name 60 surrounded by special characters, "<<" and ">>", a *wrap* attribute command string 62, and a path identifier 64. The PDL specification generated by the application program will include the graphic states of the attribute string 58. These graphic states can include the font size (i.e., 10 point), the type-font (i.e., Script) the orientation (i.e., angled upwardly at 50°) and the like.

As discussed above, and referring again to Figs. 1-4, the control task 22 will execute a PDL interpreter program to interpret the PDL specification created by the application program to generate a template bitmap 25 of the document, and to also monitor for any variable data paths defined in the PDL specification.

In the preferred embodiment, the control task 22 monitors for variable data areas defined by the PDL specification by monitoring for special characters in the text strings

defined by text commands in the PDL specification. As shown in Fig. 4, the special characters "<<" and ">>" surround the field name 60. The control task, upon identifying the special characters in the text command for the attribute string will thus know that the attribute string 58 is defining a variable data area, and is not merely defining a text string to appear on the printed page (the attribute string will not appear on the final printed page unless the control task is directed to by the job ticket file). The field name 60 surrounded by the special characters identifies the associated field name 52 present in the merge file 16. During the processing of the text command for the attribute string 58, the control task will also monitor for the *wrap* string 62 within the attribute string, which also includes the path identifier string 64 associated therewith. If found, the control task will know that a path defined in the PDL specification that has a predetermined relationship with the text command for the attribute string will be a wrapping path, where the wrapping path has the wrapping attributes defined in the job ticket file 12 for the particular group header 44 and descriptive name of a path 42 matching the path identifier string 64 set forth in the attribute string 58.

Preferably, the predetermined relationship is satisfied by the first path command to follow the text command for the attribute string in the PDL specification. This can be accomplished by using the application program to sequentially type the attribute string 58 and then draw the path 56, such that the path command will be the first path command to follow the text command in the PDL specification created by the application program. Alternatively the predetermined relationship can be satisfied by the path command that is "grouped" with the text command for the attribute string in the PDL specification. This can be accomplished by using a "GROUP" tool as provided by many application programs to group the attribute string 58 and path 56 together. It will be apparent to one of ordinary skill in the art that there are many similar predetermined relationships available between the text command for the attribute string and the path command for the wrapping path that can be established in the PDL specification, all of which fall within the scope of the present invention.

Thus, during the execution of the PDL interpreter program, the control task 22 will match the *wrap* attribute command string 62 and path identifier 64 with the group header 44 and descriptive name of the path 42 defined in the job ticket file 12. Once the attribute string 58 is identified as defining a variable data path by the control task 22, the control task will save the graphic states 23 of the attribute string 58 in memory. The control task may also create a font cache according to the graphic states 23, and store the font cache along with the graphic states in memory 24. The control task will also save the field name 60 along with the graphic states 23 in memory so that the particular graphic states can be matched to the blocks of text data in the merge file 16 under the matching field name 52, as will be described below. The merge task 28 will apply these graphic states 23 and associated font cache to the variable data 54 prior to merging and flowing the variable data into the path 56.

Once the control task 22 has identified the path as being a variable data path, and has reserved the graphic states 23 of the attribute string 58 associated with the path in memory 24, the control task 22 advances to the next data area in the PDL specification, preferably without allowing the attribute string data or the path to be added to the template display list 25 stored in memory 26. And once the PDL interpreter program has completed interpreting the PDL specification, the control task 22 then passes authority to the merge task 28.

The merge task 28 first accesses a set of the saved graphic states 23 and identifies the field name 60 associated with these graphic states. The merge task 28 then accesses the merge file 16 and searches the merge file for a field name 52 matching the field name 60 associated with the graphic states. The merge task then accesses a variable data block 54 associated with the field name 52 and then generates rendering commands for the variable data block according to the graphic states 23, the predefined flow rule 48 and the boundary of the path 56. The predefined flow rule 48 may or may not be defined by the job ticket file 12. Accordingly, when the rendering command is executed the bit map data defined by the rendering command will flow within the path 56 according to a predefined flow rule.

As shown in Fig. 11, and as illustrated in Figs. 5-10, a method for merging and flowing the variable text data into the path 56 is as follows: as indicated in step 100 and illustrated in Fig. 5, preferably the control task will first "flatten" the path, which involves breaking the complex path 56 (which may contain ellipses and curves) into a series of simple straight lines 64 (i.e., converting the path into a series of "move to" and "line to" commands). Each straight line 64 will comprise a particular portion of a boundary 65, into which the variable data is to be positioned. Alternatively, it is within the scope of the present invention to have the path 56 itself define the boundary into which the variable data is to be positioned. As will be described below, the extent of the boundary may also be defined, in part, by the designation of margins, or the creation of additional paths, etc. As indicated in step 102 and as also illustrated in Fig. 5, a horizontal axis 67 of a coordinate system 69 will be aligned with the attribute string 58. As indicated in step 104 and as illustrated in Fig. 6, a new equivalent boundary 65' is created, whose coordinates are those of the original boundary 65, but rotated into the same coordinate system 69 as the attribute string 58 (for example, as shown in Fig. 5, the attribute string 58 is rotated a negative 50° in the document, and therefore, in Fig. 6 the boundary 65' is rotated by a positive 50°).

As indicated in step 106, the stored graphic states 23 (e.g., font-type and point size) are applied to a variable data block 54 to be merged into the boundary 65' and to calculate the dimensions of a plurality of word bitmaps, the word bitmaps being defined by a collection of characters separated from the rest of the data by white space characters (e.g., a space, tab, new line, etc.). The dimensions of paragraphs can be calculated by defining a paragraph as a collection of word bitmaps separated from other paragraphs by "new line" characters. Assuming that the text flow direction will be from top to bottom and left to right, as indicated in step 108 and as illustrated in Figs. 7 and 8, the "top" or highest point 66 of the path 65' is determined and a top margin 68 is applied to the boundary 65' by measuring a distance downward from the highest point 66 of the boundary. The top margin 68 can be pre-defined, defined in the job ticket file 12, or by any other sufficient means.

As indicated in step 110 and illustrated in Figs. 7 and 8, a rectangular insertion area 70 is defined, having a vertical height corresponding to the calculated vertical height of the bitmap representation of the first word (the point size of the text) to be flowed into the boundary 65', and having a top horizontal border 72 abutting the top margin 68. As indicated in step 112, this insertion area 70 will be overlayed onto the entire boundary 65' at that present vertical level to establish at least one intersection point 74. As indicated in step 114, only those areas between adjacent intersection points 74 will be considered valid candidates for receiving the bitmap representations of the text data. If there are more than two intersection points present within the insertion area, then the particular flow rule being utilized will determine between which of the intersection points that the bitmap representations of the text data will be inserted. As illustrated in Figs. 7 and 8, when only two intersection points are established, the bitmap representations of the text data will typically be inserted therebetween.

Once two adjacent intersection points 74 are determined to be candidates for receiving bitmap representations of the text data, as indicated in step 116 and illustrated in Fig. 8, left and right margins will then be measured inwardly from each of the intersection points 74 to define left and right borders 77 within the rectangular insertion area 70. Between the left and right borders 77, therefore, is defined a text placement area 78 for merging the bitmap representations of the text data therein. The left and right margins 76 can be pre-defined, defined in the job ticket file 12, or determined by any other sufficient means.

As indicated in step 118, the rendering commands to create the bitmap representations of a word of the text data as merged into the text placement area are created and added to the display list 25, depending upon whether the calculated width of the bitmap is equal to or less than the available width calculated to remain in the text placement area. The rendering commands will define the proper orientation of the bitmap representation of the word rotated back into the original orientation of the attribute string.

As illustrated in Fig. 8, in the first text placement area 78, bitmap representations of the words "in" and "a" were able to fit therewithin, however, the bitmap representation of the word "world" was too wide for the remaining width. Accordingly, in the final merged bitmap only the bitmaps representing the words "in" and "a" will be rendered into the first text placement area 78. If no word bitmaps are capable of fitting within the text placement area, then the area is left blank.

As indicated in step 120 and illustrated in Fig. 8, a line-spacing 79 is measured below the present insertion area and then the next rectangular insertion area 80 is created and overlayed onto the boundary 65' below the line-spacing 79 in the same manner as defined above for the first rectangular insertion area 70. As indicated in step 122, if the new insertion area extends below the lowest point of the boundary 65' (or below the bottom margin) or if there are no more words to insert, then the merging process for this particular boundary and text block is finished as shown in step 124. If the insertion area does not extend below the lowest point of the boundary and there are more bitmaps representing words to insert, then the process returns to step 114, described above. Essentially, steps 114-122 will be repeated thereafter until step 124 is reached. As illustrated in Fig. 8, bitmaps representing the words "world" and "of" were able to be rendered into the second rectangular insertion area 80 and bitmaps representing the words "interactive," "media" and "and" were able to rendered into third rectangular insertion area 82.

Subsequent to step 122, the merge task will then search for additional variable data areas or variable data paths in which to merge variable data blocks. If no more of such variable data areas or variable data paths exist for the particular document, then the merged display list 30 is transferred to the rendering task 32, as described above, to generate the bitmap bands 34 for printing. Fig. 9 illustrates the entire block of text 54 from the merge file 16 formatted according to the above process and merged into the path 56 to create a first finished document 84. Fig. 10 illustrates the appearance of the next block of text 54' from the merge file 16 formatted according to the above process and merged into the path 56 to create a second finished document 86.

Preferably, in the above step 118, the height of the rectangular insertion area is determined by the dimensions calculated for the first word bitmap. And if, for whatever reason, a next word bitmap is calculated to be higher than the first or previous word bitmap, and higher than all other word bitmaps inserted thus far into a particular text placement area, then the entire rectangular insertion area is thrown out, and steps 116 and 118 are repeated again for the higher rectangular insertion area generated according to this higher word bitmap.

As discussed above, a number fill rules are available for flowing the word bitmaps into the boundary. Accordingly, the merge task can mark the path intersections 74 as "positive," "negative" or "neutral" based upon whether the path enters and leaves from the top or the bottom of the insertion area, or whether it enters and exits the insertion area from the same direction. All of the available fill rules will be apparent to one of ordinary skill in the art, and are thus within the scope of the present invention.

As discussed above, text flowing into the boundary 65' will continue until it is determined that there are no more word bitmaps to flow into the boundary or until it is determined that there is no more text areas available to flow the word bitmaps into. In the case of the latter, it is within the scope of the invention to define a path as an "overflow" path for continuing the flowing of the text therein, until this overflow path runs out of room. This overflowing process can continue until once again it is determined that there are no more text areas to flow text into. Text can also flowed into more than one path at a time.

For illustration, as shown in Fig. 12, if the job ticket file defines the number of flow paths as two, and the two flow paths are the circle and square paths, designated as numerals 88 and 90, respectively; then the two paths essentially comprise one boundary, and text will flow directly from the circle path 88 into the square path 90. Note that the 2nd through 8th lines of text flow from the circle path 88 directly into the square path 90. But when the text reaches the end of the square path 90, the flowing operation stops because the area within the two flow paths has been used up.

Accordingly, as illustrated in Fig. 13, if an "overflow path" is designated in the job ticket file to be the triangle path 92, the text flowing will continue into the triangle path 92 until there is no more text to be merged or until the path runs out of additional room.

The operation of the present invention is illustrated by way of a second example as shown in Figs. 14-18. This second example illustrates the use of the present invention in constructing a book having variable text and picture placement, where a character name presented in the book may also be customized. Once customized, the text and pictures will flow into the pages of the book regardless of the size differences between the substituted character names. For example, if a substituted character name is substantially longer or shorter than the original character name in the text, the text and pictures will flow throughout the book such that no noticeable gaps or overflows are detectable. In order to perform such a task, the present invention allows a plurality of different merge files or data items to be flowed into a single path; the present invention allows text to flow around pictures that are inserted into the path; and by utilizing special delimiters within the merge file, the merged task is able to recognize points in the merge data where the graphic states to be applied to such merge data are to be changed in accordance with a next attribute string in the PDL Specification. This is all explained in detail as follows:

As illustrated in Fig. 14, the job ticket file 12 contains a group header, "[PageDescriptionLanguageFile]" 126 specifying the file path(s) defined thereunder as determining the locations of the 'template' PDL files. In the present example, the template PDL file path 128 defines the location of the template (PostScript) file "jungle.ps" as shown graphically in Fig. 16. Next, the job ticket file 12 lists a group header, "[MergeFiles]" 144 specifying the labels ("names" and "rikkitxt") of the merge files to be accessed by the merge task. The group header "[names]" 146 is thereafter defined in the job ticket file to provide information about the merge file "names.txt" located in the file path 130. As indicated by the definitions following the group header "[names]" 146, this merge file is a delimited merge file where the record delimiters are '/n' and the field delimiters are '|'. In this merge file, the definition DoGlobalSubstitution

is set to FALSE, which indicates that substitutions of the text within the merge file are not to be performed by the merge task during the merging operation. The group header "[rikkitxt]" 148 provides information about the merge file "rikki.txt" located in the file path 132. The MergeType definition is set to "field", which indicates that the merge file only contains a single record; and therefore requires no record delineations. The MergeHeader definition is set as NO, which indicates that the merge file will not include a merge header (because there is only one record in the merge file). As also defined under the "[rikkitxt]" group header is that the field delimiter will be '#' character, the page break delimiter will be the '~' character and the paragraph delimiter will be the '@' character. Finally, the definition DoGlobalSubstitution is set to TRUE which means the merge task is to look for text phrases within the rikki.txt merge file and replace them with variable data as defined in the job ticket file as follows.

The group header 150 "[mergefile:substitution]" establishes the global substitutions for the "rikki.txt" merge file as described above. Accordingly, within the body of the "rikki.txt" merge file, every instance of the name Mowgli is to be changed to the variable data name listed under the "name1" heading (which is present in the "names" merge file -- not shown). Furthermore, any occurrence of the name Teddy within the "rikki.txt" merge file will be replaced with the same variable data name as listed under the "name1" heading in the "names" merge file. This substitution is preferably performed by the merge text when creating bitmaps for the merge data in the "rikki.txt" merge file that are to be merged into the template defined in the "jungle.ps" file (Fig. 16).

The next group header 136 "[Wrap]" in the job ticket file 12 contains a descriptive name of a path 134 (in this case, named "path") for identifying a name of a path in the PDL file that is to have variable data flowed into it during the merge task. The group header 136 "[Wrap]" indicates that the group is defining a wrapping path. After the wrapping path "path" has been defined in the job ticket file, a next group header 138 "[path]" is thereafter defined to provide information about the wrap path, such as defining the FillRule 140 as using the even/odd rule, defining the DrawPath

definition as FALSE 142 to indicate that the path is not to be drawn. The other definable wrapping commands for the particular path "path" are described in detail in the appendix below.

Although not shown in Fig. 14, the job ticket file 12 includes attribute definitions defining the print job as a 'book' job, which directs the merge task to repeatedly access templates and flow bitmaps into the path(s) in the templates until the merge task reaches the end of the merge file.

As illustrated in Figs. 15A and 15B, the merge file "rikki.txt" 16 is a platform independent data file that contains the 'variable' data to be merged into the path defined in the PDL specification (Fig. 16). In the present example, this merge file does not contain a field name because the MergeHeader definition in the job ticket file 12 was set to NO. In the present example, the mergefile is a single data record consisting of the text of the Rikki-Tikki-Tavi story of the Jungle Book. Paragraph delimiters 154 ('@') are placed at selected points within the text to inform the merge task where to start a new paragraph during the merging operation. Field delimiters 156 ('#') are also placed in selected areas of the text to indicate to the merge task when a particular field of the merge file has ended and a next field of the merge file is to begin. The use of the field delimiters 156 will be described in greater detail below.

As illustrated in Fig. 16, the designer will utilize an application program to create a template document 157 containing a path 158 and several attribute data strings 160. As discussed above, the designer will associate the attribute data strings 160 with the path 158 by assuring that the path 158 is the first path drawn after the insertion of the attribute data strings 160 or by using a "GROUP" feature of the application program to group the attribute data strings 160 with the path 158. As also shown in Fig. 16, the template document 157 also contains static data 162 which will remain constant during every printing of the merged document. Once the template document 157 has been created, the application program will then be directed to create at PDL specification 14 of the document. Each attribute string 160 contains a field name 164 surrounded by special characters, a *wrap* attribute command string 166, and a path identifier 168 if the

attribute data is to be associated with a path. The PDL specification generated by the application program will include the graphic states of the attribute strings. For example, the graphic states for first attribute string 170 include a ***bold/italics*** font attribute and a larger point size attribute; the graphic states for second attribute string 172 include an *italics* font attribute and a smaller point size attribute than the first attribute string; the graphic states for third attribute string 178 include a standard font attribute, etc.

As discussed above, referring to Fig. 1 and Figs. 14-17, the control task 22 will execute a PDL interpreter program to interpret the PDL specification created by the application program to generate a template bit map 25 of the template document 157, and to also monitor for any variable data paths defined in the PDL specification 14 . During the execution of the PDL interpreter program, the control task 22 will match the path identifier 168 in each *wrap* attribute command string 166 with the group header 136 and descriptive name of the path 134 defined in the job ticket file 12 . Once the attribute string 166 is identified as defining a variable data path by the control task 22, the control task will save the graphic states 23 of the attribute string 166 in memory (which is preferably a stack). The control task may also create a font cache according to the graphic states 23 , and store the font cache along with the graphic states to memory 24. The control task will also link the graphic states 23 with the merge file defined by the job ticket file having a name matching the field name 164 ("rikitxt" for the first, second, third and fifth attributes strings 170, 172, 178, 182 and 190). The merge task 28 will apply these saved graphic states 23 and the associated font cache to the variable data prior to merging and flowing the variable data into the path 158.

Once the control task 22 has identified the path as being a variable data path, and it has reserved the graphic states 23 of the attribute strings 166 associated with the path in memory 24, the control task 22 advances to the next data area in the PDL specification, preferably without allowing the attribute strings or the path to be added to

the template display list 25 stored in memory 26. Once the PDL interpreter program has completed interpreting the PDL specification, the control task 22 then passes authority to the merge task 28.

The merge task 28 first accesses a first set of graphic states 23 from memory 24 and identifies the particular field name 164 associated with these graphic states. The merge task 28 then accesses the merge file 16 associated with this field name and graphic states. The merge task then accesses a variable data block associated with a first variable data block in the merge file and then generates rendering commands for the variable data block according to the graphic states 23 , the predefined flow rule 140 and the boundary of the path 158.

As illustrated in Figs. 17A-17C, a method for merging and flowing the text data in the merge file into the path 158 of the document 157 to create the variable length book is as follows. Upon initiation, the merge task will first access the saved graphic states and attributes associated with the first attribute string 170 defined in the PDL specification. As shown in Fig. 16, the field name is "rikkitxt" and the path associated with the attribute string 170 is the path 158 (because the path 158 is the first path created after the attribute string 170). Referring to Fig. 14, the merge task matches the field name "rikkitxt" 164 in the first attribute string 170 with the group header 148, and accesses the merge file identified by the path 132. As shown in Fig. 15A, a first text-block 171 is taken from the beginning of the mergefile until a first field delimiter 156a is encountered. The saved graphic states 23 associated with the first attribute string 170 are applied to this text block to create a bit map for the text block which is then flowed into the path 158 as shown by numeral 172 in Fig. 17A. Note that the attribute string 170 included an attribute command "textc," which caused the control task to add an additional text centering attribute to the saved graphic states 23 . Accordingly, the bit map 172 associated with the text string and applied graphic states is centered in the path 158. The paragraph delimiter 154 in the mergefile causes the merge task to add a line space after the insertion of the bit map 172.

Because the merge task reaches the first field delimiter 156a in the mergefile, the merge task refers back to memory to retrieve the reserved graphic states 23 attributed to the second attribute string 172. The field name 164 identified by the second attribute string 172 is "rikkitxt" as in the first attribute string 170; and therefore, the merge task will again refer to the mergefile 152 when retrieving variable data to insert into the path. It is should be apparent to those of ordinary skill in the art that the field name 164 may also refer to a different merge file and the merge task would thus access data from the different merge file. As with the first attribute string 170, the second attribute string 172 includes the additional attribute commands such as "textc" and "padjust=0." Referring again to Fig. 15A, the merge task will access the next block of data 173 between the first field delimiter 156a and the second field delimiter 156b. The merge task will then apply the graphic states 23 corresponding to the second attribute string 172 to this text data to form the bit map data block 174 to be merged into the path 158. Once this bit map block has been merged into the document, the merge task accesses the graphic states associated with the next attribute string 178 from memory.

Because the field name 164 in the third attribute string 178 is "rikkitxt" as with the first two attribute strings, the merge task will refer back to the mergefile 152 and will extract the block of data 179 after the second field delimiter 156b and before the third field delimiter 156c. The graphic states 23 associated with the third attribute string 178 will be applied to this text data to create bit map data which is merged and flowed into the path 158 according to the steps described herein. Once the merge text reaches the end of the path 158, the merge task will know to access another copy of the template from memory because a "book" attributes have been predefined in the job ticket file. The second template bit map is indicated in Fig. 17B. Note that the block of text flows beyond the path 158 of the second template bitmap shown in Fig. 17B and into the path 158 of the third template bitmap shown in Fig. 17C. Once this block 180 has been

mapped and the merge task reaches the third field delimiter 156c, the merged task refers back to the graphics states 23 in memory to obtain the graphic states associated with the fourth attribute string 182.

The field name 164 in this attribute string 182 refers to "rpic1", which is defined in the job ticket file as a bit map of a picture to be inserted at this point. Note that this attribute string also includes additional attribute commands: "textl" and "dropcap". This indicates that the picture bit map is to have left justification and is to be treated as a drop-cap character. As shown in Fig. 17C, the picture bitmap is inserted into the path 158 with left justification after the block of bit map data 180. If the drop-cap command had not been specified in the attribute string, the next block of data would be inserted at point 185 after the picture bit map. However, it is often desirable to include pictures within the text of a book and then have text appear to flow around the picture. Accordingly, the drop-cap attribute definition indicates to the merged text to treat the bit map defined in the attribute string as a drop-cap character. When the merged task sees this command, after inserting the picture 183 into the path 158 the merged task *adds* the boundary 184 of the picture to the path 158 and then moves the insertion point of the next bit map data to the beginning 186 of the picture bit map 183. However, because the boundary 184 of the picture bit map 183 has been combined with the boundary 158 of the path, the next insertion point will be at point 188 to the right of the picture bit map.

Once this step is completed, the merge task will access the graphic states 23 associated with the next attribute string 190 from memory. The field name "rikkitxt" 164 in this next attribute string 190 indicates to the merge task to access data again from the merge file 152. Referring to Fig. 15C, the next point to access data for the merge file is the block of data indicated by numeral 192, between the third and fourth field delimiters 156c, 156d. The graphic states 23 of this next attribute string 190 will be applied to this block of data 192 and the bit maps will thus be flowed into the path 158 as discussed above. This block of data 192 is the first block including a character name 'Teddy' which the job ticket directs as needing to be replaced by a variable data

name from the 'names.txt' merge file as discussed above. In the present example, the first variable name listed in the 'names.txt' merge file is "Ranen." Accordingly, merge file will replace all instances 193 within the block of data 192 where the name 'Teddy' appears with the 'Ranen' bitmaps 195 in the printed document. This process will continue until the merge task reaches the end of the mergefile 152, indicating to the merge task that the book has been created. Figs. 18A-18C illustrate the appearance of the pages of the book as prepared in the example described above.

Accordingly, the present invention provides capability of identifying particular paths defined in a page description language as data flowing paths, and provides the capability for flowing data within such paths. In addition, the present invention allows the user to specify margin, paragraph formatting, fill rules, and justification parameters on a path by path basis.

Having described the invention in detail and by reference to the drawings, it will be apparent to one of ordinary skill in the art that variations and modifications are possible without departing from the scope of the invention as defined on the following claims.

The following appendix provides a preferred compilation of text wrapping commands and parameter definitions that can be specified in the job ticket file 12. Each entry provides the particular command header, the syntax for the command, any relevant remarks for the use of the command, examples, etc. As will be apparent to one of ordinary skill in the art, it is within the scope of the present invention to include the means to provide for any of the attributes, or similar attributes, as defined in the Appendix.

APPENDIX

COMMAND HEADER = [Wrap]

A group that provides a list of tags which you create to describe the text flowing (wrap) path(s) to be used in the print job. Each tag will become a user-defined group of additional information about the wrap path.

Syntax [Wrap]
<Path Tag X.>
<Path Tag Y>
<Path Tag Z>

Remarks Optional. Each tag that appears under this [Wrap] group will optionally become a new group name in a succeeding section of the Job Ticket.

Explanation <Path Tax X>
Create a descriptive name for a wrap path used in the print job.
Note: Fields on a template that you wish to be flowed into a particular path will use a field attribute of the format:

<<fieldname>> wrap=<name>

The <name> argument of the wrap attribute must match a path tag listed in the [Wrap] group.

Example [Wrap]
Circle
Square
Triangle

COMMAND HEADER = [<Path Tag>]

A user-defined tag name for a group that provides information about the wrap path and corresponds to the descriptive tag that you create under the initial [Wrap] group.

Syntax [<Path Tag>]

Baseline Adjust =
Bottom Margin =
Clobber Path =
Draw Path =
Enforce paragraph Spacing =
Fill Rule =
Fit Last Name =
Justify =
Left Margin =
Margins =
Min Paragraph Lines =

Number Of Paths =
Overflow =
Paragraph Adjust =
Reverse Flow =
Reverse Path =
Right Margin =
Top Margin =

Remarks A separate [<Path>] group defines path information for each descriptive tag listed under the initial [Wrap] group.

If a [<Path Tag>] group is not defined for a path listed under the [Wrap] group, that path will receive the default values for all of the [<Path Tag>] elements.

Explanation [<Path Tag>]
Take the descriptive tag under the initial [Wrap] group and write it here as a group name within the brackets [ ].

Baseline Adjust =
(See the Baseline Adjust element description)

Bottom Margin =
(See the Bottom Margin element description)

Clobber Path =
(See the Clobber Path element description)

Draw Path =
(See the Draw Path element description)

Enforce Paragraph Spacing =
(See the Enforce Paragraph Spacing element description)

Fill Rule =
(See the Fill Rule element description)

Fit Last Line =
(See the Fit Last Line element description)

Justify =
(See the Justify element description)

Left Margin =
(See the Left Margin element description)

Margins =
(See the Margins element description)

MinParagraph Lines =
(See the MinParagraph Lines element description)

Number Of Paths =
(See the Number Of Paths element description)

Overflow =
(See the Overflow element description)

Paragraph Adjust =
(See the Paragraph Adjust element description)

Paragraph Indent =
(See the Paragraph Indent element description)

Reverse Flow =
(See the Reverse Flow element description)

Reverse Path =
(See the Reverse Path element description)

Right Margin =
(See the Right Margin element description)

Top Margin =
(See the Top Margin element description)

Examples [Circle]

Fill Rule	=	EvenOddRule
DrawPath	=	False
Overflow	=	Square

[Square]

FillRule	=	WindingRule
DrawPath	=	True
Overflow	=	Triangle

```
[Triangle]
        FillRule =      EvenOddRule
        DrawPath        =       False
        Overflow        =       Square

[Square]
        FillRule =      WindingRule
        DrawPath        =       True
        Overflow        =       Triangle

[Triangle]
        FillRule =      EvenOddRule
        DrawPath        =       False
```

PARAMETER = Baseline Adjust

An element that determines the adjustments made to each baseline of text drawn within the path.

Syntax Baseline Adjust = <BaseAdjustNum><Unit Type>

See Also Paragraph Adjust, Enforce Paragraph Spacing.

Remarks Optional.

By default, the process will space successive text lines at 120% of the font size. For example, a 12-point font will have the next baseline set at 14.4 points (120% x 12) from the previous baseline. The Baseline Adjust element defines an offset from this default value.

A positive Baseline Adjust value increases the space between each baseline of text (essentially, moving the next line of text down). A negative value decreases the space between each baseline of text (essentially, moving the next line of text up).

The default value for Baseline Adjust is 0.

Explanation <BaseAdjustNum>
Enter the number of units.

<Unit Type>

Optional. Enter the abbreviation to identify the unit type if the unit type for Baseline Adjust is different from the default unit type defined in the Units element. Possible values are:

cm	for centimeters
dots	for dots
ft	for feet
in	for inch (default value)
mm	for millimeter
pts	for points

Example BaselineAdjust = 1pt

PARAMETER = Bottom Margin

An element that specifies the distance from the bottom of the path at which to stop flowing text.

Syntax BottomMargin = <BottomMarginNum><Unit Type>

See Also Margins, Overflow.

Remarks Options.

NOTE: A non-zero value for the BottomMargin element overrides (for the bottom margin only) the value set in the Margins elements.

For example, if Margins = 1in and BottomMargin = 2in, the path will have 1-inch margins on the top, left, and right sides but will have a 2-inch margin on the bottom side.

The default value for Bottom Margin is 0.

Explanation <BottomMarginNum>
Enter the number of units.

<UnitType>
Optional. Enter the abbreviation to identify the unit type if the unit type for Bottom Margin is different from the default unit type defined in the Units element. Possible values are:

cm	for centimeters
dots	for dots
ft	for feet
in	for inch (default value)
mm	for millimeter
pts	for points

Example BottomMargin = 3mm

PARAMETER = Clobber Path

An element that specifies if two adjacent ON areas separated by a path segment are treated as one area when determining text flow.

Syntax ClobberPath = [True/False]

See Also FillRule

Remarks Optional

This element affects the way in which text is flowed in adjacent ON areas. It applies only to paths defined with FillRule = WindingRule.

If ClobberPath is set to True, text is flowed across the two adjacent ON areas as if they were one area. In this case, only the "outer" margins of the combined areas would be recognized. Text flow would be continuous across the "inner" margins where the path segment intersects the adjacent areas.

If ClobberPath is set to False, text is flowed separately into each area.

The default value of ClobberPath is True.

Explanation {True/False}
If two adjacent ON areas are to be treated as one area, type True.

If two adjacent ON areas are to be maintained separately, type False.

Example ClobberPath = False

PARAMETER = DrawPath

An element that determines if the wrap path is actually drawn on the template.

Syntax DrawPath = {True/False}

Remarks Optional.

The default value for DrawPath is True.

Explanation {True/False}
If the wrap path is to be drawn on the template, type True.

If the wrap path is NOT to be drawn on the template, type False.

Example DrawPath = False

PARAMETER = EnforceParagraphSpacing

An element that determines if the next paragraph will always start at a distance of the ParagraphAdjust value from any previous paragraphs that were set.

Syntax EnforceParagraphSpacing = {True/False}

See Also BaselineAdjust, ParagraphAdjust.

Remarks Optional.

If the text flowed into your path contains blank paragraphs, this element determines how the blank paragraphs are to be handled.

If you want your next paragraph to start at a distance of the ParagraphAdjust value from your previous text paragraph (thereby, "skipping" any blank paragraphs and permitting text to continue to flow), set the EnforceParagraphSpacing value to True.

If you want the blank paragraphs to be allotted the appropriate space defined in ParagraphAdjust, set the EnforceParagraphSpacing value to False.

The default value for EnforceParagraphSpacing is False.

Explanation {True/False}

If the next non-blank paragraph should start at a distance of the ParagraphAdjust value from any previous paragraphs that were set, type True.

If blank paragraphs are to be allocated their appropriate paragraph space, type False.

Example EnforceParagraphSpacing = True

PARAMETER = FillRule

An element that provides the rules used to determine which areas of the path should have text flowed into them and which areas should be blank.

Syntax FillRule = {WindingRule/EvenOddRule}

See Also ClobberPath, ReversePath.

Remarks Optional.

Text is flowed into an area enclosed by ("inside") the current path. If a path is simple, it is clear which areas are inside the path. However, if a path is complex (for example, intersecting itself or having one subpath that encloses another), it is not as apparent which areas are inside. One of two fill rules will be used to determine which areas lie inside a path.

The FillRule element defines if the non-zero winding rule (WindingRule) or the even-odd rule (EvenOddRule) will be used for the current path.

The *non-zero winding rule* determines whether a given area along the proposed flow line is inside the path (and thus receives text) by examining the places where a path segment crosses the flow line. Path segments that cross (intersect) the flow line from top to bottom are given a direction of 1. Path segments that cross (intersect) the flow line from bottom to top are given a direction of -1. Path segments that do not fully cross the flow line (for example, entering and exiting the top of the flow line) are given a direction of zero.

An on-going sum of all crossings is calculated from left to right. If the sum of all crossings to that point is zero, the area (immediately to the right) is outside the path. If the sum is non-zero, the area is inside the path and will receive text.

The *even-odd rule* determines whether a given area long the proposed flow line is inside the path (and thus receives text) by counting the number of times a path segment crosses the flow line. Path segments that fully cross (intersect) the flow line are given a score of 1. Path segments that do not fully cross the flow line are given a score of zero.
An on-going sum of all crossings is calculated from left to right. If the sum of all crossings to that point is even, the area (immediately to the right) is outside the path. If the sum is odd, the area is inside the path and will receive text.

The default value for FillRule is WindingRule.

Explanation {Winding Rule/EvenOddRule}
If the winding rule will determine which areas lie inside a path, type WindingRule.

If the even-odd rule will determine which areas lie inside a path, type EvenOddRule.

Example FillRule = EvenOddRule

PARAMETER = FitLastLine
An element that determines if the Fit justification rule is applied to the last line of every paragraph.

Syntax FitLastLine = {True/False}

See Also Justify

Remarks Optional.

The FitLastLine element applies only to paths defined with Justify = Fit.

If FitLastLine is set to True, the text on the last line will be forced to fit flush on the left and the right. Since the last line of a paragraph

may often contain less text than the other lines in a paragraph, this justification will often result in more white space between text on the last line.

The default value for FitLastLine is False.

Explanation {True/False}
If the last line of every paragraph should be aligned ab both the left side and the right side of the path, type True.

If the last line of every paragraph should not be forced to fit flush left and flush right, type False.

Example FitLastLine = False

PARAMETER = Justify

An element that specifies the type of justification (horizontal alignment) to be applied to each line of text drawn in the path.

Syntax Justify = <JustifyRule>

See Also FitLastLine

Remarks Optional.

The default value for Justify is Left.

Explanation <JustifyRule>
Enter the type of justification (horizontal alignment) to be applied to each line of text drawn in the path. Possible values are:

Left (Default value) Text is aligned from the left side of the path.

Right Text is aligned from the right side of the path.

Center Text is centered between the left side and right side of the path.

Fit Text is aligned at both the left side and right side of the path.

Example Justify = Center

PARAMETER = LeftMargin

An element that specifies the distance from the left side of the path at which to start flowing text.

Syntax LeftMargin = <LeftMarginNum><UnitType>

See Also Margins

Remarks Optional.

NOTE: A non-zero value for the LeftMargin element overrides (for the left margin only) the value set in the Margins elements.

For example, if Margins = 1in and LeftMargin = 2in, the path will have 1-inch margins on the bottom, top, and right sides but will have a 2-inch margin on the left side.

A default value for LeftMargin is 0.

Explanation <LeftMarginNum>
Enter the number of units.

<UnitType>
Optional. Enter the abbreviation to identify the unit type if the unit type for LeftMargin is different from the default unit type defined in the Units element. Possible values are:

cm	for centimeters
dots	for dots
ft	for feet
in	for inch (default value)
mm	for millimeter
pts	for points

Example LeftMargin = 5 mm

PARAMETER = Margins

An element that specifies the same text margins for all four sides of the path (top, bottom, left, and right).

Syntax Margins = <MarginsNum><Unit Type>

See Also BottomMargin, LeftMargin, Right Margin, TopMargin

Remarks Optional.

Note: The value for the Margins element will be overridden on an individual margin basis by any non-zero value defined for the other specific margin attributes (BottomMargin, LeftMargin, RightMargin, and TopMargin).

For example, if Margins = 1in and TopMargin = 2in, the path will have 1-inch margins on the bottom, left, and right sides but will have a 2-inch margin on the top.

The default value for Margins is 0.

Explanation <MarginsNum>
Enter the number of units.

<UnitType>
Optional. Enter the abbreviation to identify the unit type if the unit type for Margins is different from the default unit type defined in the Units element. Possible values are:

cm	for centimeters
dots	for dots
ft	for feet
in	for inch (default value)
mm	for millimeter
pts	for points

Example Margins = 6 mm

PARAMETER = MinParagraphLines

An element that specifies the minimum number of lines of a paragraph to be set before the paragraph is allowed to be split between path areas.

Syntax MinParagraphLines = <MinLinesNum>

See Also NumberOfPaths, Overflow.

Remarks Optional.

If the minimum number of lines of a paragraph defined here cannot be set consecutively in a path area, the entire paragraph will be moved down to the next scanline that allows the specified number of lines to be set consecutively.

The default value for MinParagraphLines is 1.

Explanation <MinLinesNum>
Enter the integer representing the minimum number of lines of a paragraph to be set before splitting between path areas is permitted.

Example MinParagraphLines = 2

PARAMETER = NumberOfPaths

An element that determines how many postscript paths on the template are concatenated and treated as one path.

Syntax NumberOfPaths = <PathsNum>

See Also MinParagraphLines, Overflow.

Remarks Optional.

This element is used to combine multiple paths drawn on the template and to treat them as a single path. The path to be combined will be determined by the order in which they were drawn.

The default value for NumberOfPaths is 1.

Explanation <PathsNum>
Enter the integer representing the number of paths to be combined.

Example NumberOfPaths = 2

Illustration See Fig. 12 and corresponding description above

PARAMETER = Overflow

An element that specifies the name (tag) of the wrap path that will receive overflow text from the current wrap path being described.

Syntax Overflow = <PathTag>

See Also MinParagraph Lines, NumberOfPaths.

Remarks Optional.

This element defines the use of an overflow feature. When overflow is available, if the current path has no more space into which text can flow, the text will continue to flow into the path named in this element.

NOTE: If the Overflow element references a wrap path that is not named under the [Wrap] group, the print job will be aborted.

If the Overflow element is not defined, the system will assume that no overflow will occur for the current path being described. Therefore, text will flow into the current path until it is filled. No overflow text will be printed.

Explanation <PathTag>
Enter the descriptive tag of the path into which overflow text from the current path will flow. This value should correspond to a descriptive tag that you created under the initial [Wrap] group.

Example Overflow = Square

Illustration See Fig. 13 and the corresponding description above.

PARAMETER = ParagraphAdjust

An element that determines the distance to adjust the baseline for the start of the next paragraph within the path.

Syntax ParagraphAdjust = <ParagraphadjustNum><Unit Type>

See Also BaselineAdjust, Enforce Paragraph Spacing.

Remarks Optional.

The ParagraphAdjust value overrides the Baseline Adjust value only for the first baseline of text in each paragraph.

A position ParagraphAdjust value increases the vertical space between the last baseline of text in each paragraph and the start of the next paragraph (essentially, moving the start of the next paragraph down). A negative value decreases the vertical space between the last baseline of text in each paragraph and the start of the next paragraph (essentially, moving the start of the next paragraph up).

The default value for ParagraphAdjust is 0.

Explanation <ParagraphadjustNum>
Enter the number of units.

<UnitType>
Optional. Enter the abbreviation to identify the unit type if the unit type for ParagraphAdjust is different from the default unit type defined in the Units element. Possible values are:

cm	for centimeters
dots	for dots
ft	for feet
in	for inch (default value)
mm	for millimeter
pts	for points

Example ParagraphAdjust = 6pts

PARAMETER = ParagraphIndent

An element that specifies the indentation from the left margin for the first line of every paragraph in the path.

Syntax ParagraphIndent = <ParagraphIndentNum><UnitType>

See Also ParagraphAdjust

Remarks Optional.

The default value for ParagraphIndent is 0.

Explanation <ParagraphIndentNum>
Enter the number of units.

<UnitType>
Optional. Enter the abbreviation to identify the unit type if the unit type for ParagraphIndent is different from the default unit type defined in the Units element. Possible values are:

cm	for centimeters
dots	for dots
ft	for feet
in	for inch (default value)
mm	for millimeter
pts	for points

Example ParagraphIndent = .5in

PARAMETER = ReverseFlow

An element that determines if the text will be flowed from bottom to top in the current path.

Syntax ReverseFlow = {True/False}

See Also FillRule

Remarks Optional.

The default value for ReverseFlow is False.

Explanation {True/False}
If the text will be flowed from bottom to top, type True.

If the text will be flowed from top to bottom, type False.

Example ReverseFlow = True

PARAMETER = ReversePath

An element that determines if the ON/OFF designations for areas in the path will be reversed.

Syntax ReversePath = {True/False}

See Also FillRule

Remarks Optional.

The ReversePath element applies only to paths defined with FillRule = EvenOddRule.
If ReversePath is set for True, the areas originally marked as ON based on the EvenOddRule calculation will be set to OFF and the areas originally marked as OFF based on the EvenOddRule calculation will be set to ON.

If ReversePath is set to False, the EvenOddRule calculations will be retained.

The default value for ReversePath is False.

Explanation {True/False}
If the ON/OFF designations for areas in the path will be reversed, type True.

If the ON/OFF designations for areas in the path will be retained, type False.

Example ReversePath = True

PARAMETER = RightMargin

An element that specifies the distance from the side of the path at which to stop flowing test.

Syntax RightMargin = <RightMarginNum><UnitType>

See Also Margins

Remarks Optional.

NOTE: A non-zero value for the RightMargin element overrides (for the right margin only) the value set in the Margins element.

For example, if Margins = 1in and RightMargin = 2in, the path will have 1-inch margins on the bottom, top, and left sides but will have a 2-inch margin on the right side.

The default value for RightMargin is 0.

Explanation <RightMarginNum>
Enter the number of units.

<UnitType>
Optional. Enter the abbreviation to identify the unit type if the unit type for RightMargin is different from the default unit type defined in the Units element. Possible values are:

cm	for centimeters
dots	for dots
ft	for feet
in	for inch (default value)
mm	for millimeter
pts	for points

Example RightMargin = 5mm

PARAMETER = TopMargin

An element that specifies the distance from the top of the path at which to start flowing text.

Syntax TopMargin = <TopMarginNum><UnitType>

See Also Margins

Remarks Optional.

NOTE: A non-zero value for the TopMargin element overrides (for the top margin only) the value set in the Margins element.

For example, if Margins = 1in and TopMargin = 2in, the path will have 1-inch margins on the bottom, left, and right sides but will have a 2-inch margin on the top side.

The default value for TopMargin is 0.

Explanation <TopMarginNum>
Enter the number of units.

<UnitType>
Optional. Enter the abbreviation to identify the unit type if the unit type for TopMargin is different from the default unit type defined in the Units element. Possible values are:

cm	for centimeters
dots	for dots
ft	for feet
in	for inch (default value)
mm	for millimeter
pts	for points

Example TopMargin = .25in

What is claimed is: